https://doi.org/10.48009/1_iis_2008_129-137

SOA IN THE CONTEXT OF A COMPARISON OF DISTRIBUTED COMPUTING ARCHITECTURES AND THE IS CURRICULUM

David Wood, Robert Morris University, wood@rmu.edu
Frederick Kohun, Robert Morris University, kohun@rmu.edu
Joseph Packy Laverty, Robert Morris University, laverty@rmu.edu

ABSTRACT

This paper provides an overview of the SOA methodology through the introduction and review of antecedent modular programming paradigms and technologies. What sets SOA apart as an analysis and design methodology is the explicit inclusion of business process redesign as part of the object-oriented approach to systems development. While the Systems Development Life Cycle is a mainstay of IS curricula, the SOA methodology including business process redesign, is not as common. With the advent of web technologies and incorporation of object oriented design, more modern IS has reduced the inclusion of business analysis. This paper demonstrates that what is currently included in a typical technical IS curriculum can be integrated and taught in the context of the SOA methodology

Keywords: Service Oriented Architecture, SOA, Systems Development Life Cycle, Object Oriented Programming, Web Services, Distributed Computing, IS Curriculum.

INTRODUCTION

The understanding and comparison of various distributed computing technologies is of strategic importance as industry attempts to reuse legacy applications and lower the cost of software development. This necessarily includes the integration of business system applications between the organization and external parties. Service Oriented Architecture (SOA) provides a viable software development architecture to fulfill this need. To better understand the nature and scope of SOA challenges to the IS curriculum this paper discusses the evolution of programming architectures from structured and modular programming to various distributed computing models. Comparing SOA and Web Services to alternative Distributed Computing architectures is necessary before one can recommend IS curriculum changes in Programming Languages, Operating System, System Analysis and Project Management.

Demands to keep the IS curriculum current has always been a challenging task. The increased role of SOA in distributed application may have extensive implications on the overall IS curriculum. SOA provides the software framework for creating and using distributed application services whose functions are loosely coupled with the operating systems and programming languages underlying the applications [12]. SOA provides the theoretical foundation of today's Web Services Architecture [9,11]. SOA and Web Services can be viewed as alternative software architectures to remote procedure call technologies such as CORBA, RMI, and DCOM, and HTTP-like transactional architectures, e.g. Servlets, JSP, ASP and CGI.

REMOTE PROCEDURE CALL TECHNOLOGIES

Remote Procedure Call (RPC) technologies are request-reply middleware protocols that enable a client process to execute processes or subprocesses on a server without the programmer coding the details of the remote call. The calling program requests the services of a remote application by sending a message. On receipt of the message, the server process processes the request and sends a reply message. The format and structure of the message is automatically determined by a shared Interface Description Language (IDL). The process used by the client formatting the message using the IDL is called marshalling. The server disassembles, or unmarshalls, the message to determine the methods and parameters to be executed [3, p139].

Consider the following example of a Remote Procedure Call: A corporate subsidiary needs to prepare various financial reports, transactions and analyses. The corporate subsidiary needs to include cost allocation of the parent corporation in this report. The format and process used to allocate the parent corporation's overhead varies in terms of methodology and parameter passing. A client's application could incorporate a Remote Procedure Call to the parent corporation server's cost allocation application using the information to format and structure a message using the shared IDL.

The Common Object Request Broker Architecture (CORBA) is a distributed computing standard defined by the Object Management Group (OMG). CORBA enables software components written in multiple object-oriented computer languages to be executed though a variety of local and remote

operating systems [3 p670]. Sun Micro System Remote Method Invocation extends RPC to the Java programming languages. CORBA also supports RMI.

The Distributed Component Object Model (DCOM) is a proprietary Microsoft technology which extended the Microsoft Component Object Model (COM) to include remote procedure calls. DCOM was integrated into the.NET framework. DCOM is RPC-like. DCOM messages are based on the Distributed Computing Environment/ Remote Procedure Call (DCE/RPC) middleware protocol.

HTTP-like transactional distributed architectures are based on web browser/web server model. An HTML form is submitted to a remote web server using the HTTP protocol. The HTTP header specifies the name of a server application and includes parameters to be passed to the server process. The web server accepts the HTTP message and calls and passes the message to server application using a mechanism called the Common Gateway Interface (CGI). The application executing on the remote server processes the request and generates a dynamic web page response. Historically, CGI applications were written in an operating system scripting language such as Perl. Other server-side programming languages that use the CGI interface are Java Servlets, PHP and ASP.

The Stub/Skelton RPC architecture is the process of any client application calling any remote application. On the other hand, "HTTP-like transactions" architecture is the process of a web server accepting a HTML form submission via the HTTP protocol then using the CGI interface to call a server application. The HTTP-like distributed architecture has been most commonly used in web-based applications. Stub/Skelton distributed Architectures may interface different types of client and server applications.

CONTRIBUTIONS OF STRUCTURED PROGRAMMING PARADIGMS TO SOA

Historically, there have been many programming paradigms based on the technology and languages available. First generation or machine code languages required one paradigm. The introduction of second generation assembly languages encouraged another paradigm for program development. Third generation languages separated data from code, and after virtual memory was available, encouraged the structured programming methodologies. Database and Distributed Programming fomented still other approaches to program development. Object-oriented programming languages encourage other paradigms. Each shift in Programming Paradigms has had implications on the choice of a software development model, e.g., Waterfall, Unified Process, Agile and Extreme Programming.

Structured programming principles are a subset of procedural programming. These principles were popular in the software development of many third-generation applications. Many Structured Programming Principles, such as Top-down design, Basic Control Structures, and Modular Design are still widely accepted today. The principle of Top-down design started analysis of problems at a broad level and then continued to a more specific level. (Kendall 2002 p245) Top-down design inherently assumed that some level detail must be understood before coding could begin and that software testing should be delayed until achieved an adequate threshold of modules or stubs had been coded. Structured programming paradigms stated that any program can be expressed in one of three control structures: sequence, selection and iteration and also encouraged the removal or reducing reliance on the GOTO statement and replacing it with a program call [4].

Perhaps, the greatest contribution of Structured Programming Principles to SOA has been the evolutionary application of modular programming concepts. Original modular design concepts were developed within the scope of a single application and one program language, which were executed on one computer system. All modular design and development concepts prescribe the subdivision of a program, application or system into a smaller logical, manageable portions, or modules. Modules provide for the separation of implementation (program/business logic) and the interface. Advantages of modular design included: a) easier to code, test and debug since each module is virtually self-contained, b) easier to develop, maintain and reuse, c) easier to understand since a module should be designed to focus on one purpose [8 p756]. The effectiveness of modular designs can be classified by their internal consistency, e.g., cohesion, and its relationship between other modules, e.g., coupling.

Cohesion within a module refers to the extent to which a module performs a single function [5, p27]. The degree of cohesion can be classified on a scale from Coincidental Cohesion (Lowest Cohesion) to Functional Cohesion (Highest Cohesion). In a Function Cohesion module all parts focus on a signal well-defined requirement. High cohesive modules are easier to understand, develop, test and re-use.

Coupling represents the degree of the interrelatedness and interdependencies between two or more modules. The degree of coupling has been historically classified on a scale from Low Coupled (Message

coupling) to Tightly Coupling (Data Coupling). Modules that were designed with low coupling were associated with high cohesion (implementation) and parameter/message interfaces. With low coupling, a change in one module did not require a change in the implementation of another module, easy to re-use, and understand.

Early modular designs created interfaces based on Data Coupling, or the parameter-passing interface [5, p507]. Third generation languages such as Fortran, Cobol and C, can contain modules that use parameter interface methods considered to be tightly coupled. While the internal code of each module was independent, knowledge and stability of the parameter interface was difficult to change. The rules of parameter interfaces between the Calling module and Called module must agree in the number, sequence, semantics and data type of parameters. If a module was designed to calculate the amount of sales tax to be changed by state, both the taxable amount and state abbreviation must be included in the interface. One problem on data-coupled interfaces is which parameter was listed first?

Tightly coupled modules and components frequently must share the same programming language and platform. Tightly coupled components tend to make maintenance and reuse much more difficult because a change in one component automatically means changes in others.

CONTRIBUTIONS OF OBJECT-ORIENTED PARADIGMS TO SOA

Acceptance of modular programming principles for third generation language applications was optional. The emergence of Object-oriented programming (OOP) languages made modular programming principles a prerequisite. The theoretical concept of a program module now had a concrete program structure called a class. A class is the basic unit of an object-oriented, which contains data (attributes) and operations (methods) [10]. While a class is an abstract structure, an object, or an instance of a class can be used in a program. Although OOPs do not directly enforce the level of cohesion or coupling, reusing existing modules is one of the cornerstones of object-oriented languages.

Object-oriented program languages added new features that would enhance the effectiveness of modular programming. Object inheritance enables the programmer to improve or customized an existing module by inheriting the features of an existing class. A lower level of coupling between modules was enhanced by features such as polymorphism, encapsulation, and messages. Encapsulation hid the details of the operations that manipulated the object. Polymorphism allows the programmer to define the same function with multiple data types as parameters. To use an object all a programmer would need to know was the methods provided by that class [10, p336]. While D.L. Parnas in 1972 citied the information hiding features of modular designs [13], it was not until the emergence of object-oriented languages that feature was directly implemented.

Probably the most important impact of object-oriented programming languages was the introduction of a new form of information sharing called message coupling. Message coupling permitted the use of a published interface to exchange parameter-less messages. An objective of message coupling interfaces is to provide intercommunication modules written in different programming languages executing under different operating system environments. Message coupling is most frequently used in distributive computing environments and relies on an Interface Description Language (IDL). Interface Description Languages are designed to allow objects written in different programming languages to invoke (call) each other, even on remote systems [3, 16].

CONTRIBUTIONS OF DISTRIBUTED COMPUTING PARADIGMS TO SOA

Both legacy and object-oriented programming languages have been able to able to invoke procedures and methods in the same local machine. Distributed computing is a collection of cooperating applications that invoke modules often across one or more computer systems or locations was defined as or distributed processing. [2, p516-17]. Most distributed applications can be generically classified as Remote Procedure Calls (RPC), Remote Method Invocation (RMI), and Event-based Programming [3, p166].

Remote Procedure Call model, the earliest and best-know distributed model, served as the foundation for the N-Tier Client-Server model. The Three-Tier Client-Server Model separated application responsibilities into a) a View layer which requested services and formatted a response; b) a Business Logic Layer which implements the rules and procedures, and a Data Layer which managed and accessed stored data [2, p518]. The Client-Server model is the dominant distributed computing model used by the Internet and may be used by either legacy or object-oriented applications.

Remote Method Invocation is an extension of the client-server model to remote invocation to methods within objects in different processes to communicate

with one another. While client-server architectures are based on a Request/Reply Protocol, the Event-based model allows objects to receive notification from other objects (without a request) in which they have a registered interest [3, p166].

While IS curriculums have clearly addressed the interdependence of modules within a local process, i.e., modular coupling, the emphasis of the interdependence or interoperability in distributed applications is lacking. Many distributed applications discussions do provide lip-service to the role of Middleware in Client-Service Architectures focusing on low level file and print sharing applications. Middleware is a wide-ranging system software category that "glues" together connection and communication requirements of a client-server architecture [2, p518]. Middleware contain the basic building blocks of process and message passing. While specificity between the definitions of Middleware may vary there is common agreement that the goal of Middleware should provide location transparency, i.e., the client process need not know the location of the server process [3, p166].

There is more that needs to be addressed than simply distributing application modules across a Network and Location Transparency. No longer are application modules limited to one application, using one programming language and computer. In SOA, a program module is now seen as a software component or service. Comparisons between the degree of interoperability between component-based architectures in a distributed applications is a very important consideration not only for industry, but also for IS curricula.

A component is an independent executable software module that has a unique identifier with a well known interface [2, p518]. The technical interoperability of distributed components have become increasingly important. The proposed ISO/IEC 2382-7:2000 draft states that interoperability is "the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units"(ISOIECJTC1 2003). Expanding the degree of interoperability between software components to be classified "non-context specific"(Szyperski 2002) becomes problematic.

For example, Interface Description Languages (IDLs) are designed to allow objects written in different languages to invoke each other. IDLs provide the foundation of interoperability between programming languages. Popular IDLs include CORBA, DCOM, J2EE and XML/SOAP. Simply communicating between distributed components does not mean that these components can intercommunicate among a variety of operating systems. Hence, a discussion and comparison of interoperability software architectures is important. Interoperable software application frameworks can be classified as Stub/skeleton based RPC, HTTP-like Transactional Architectures, and SOA (Web Services).

CONTRIBUTIONS OF STUB/SKELETON RPC SOFTWARE ARCHITECTURES TO SOA

Stub/skeleton based architectures include CORBA, Java RMI and Microsoft's DCOM. In this architecture a client application, e.g. an invoice program, needs to interface with server application, e.g., an inventory control program. In order to use these architectures developers are required to build a client interface (stub) and a server interface. CORBA is language neutral architecture which means that the invoice program and the inventory control program may be written in two different languages. Developers would use a code generator to generate a language specific IDL interface.

Microsoft's DCOM architecture operates in a similar manner, but can only use languages that support the .NET framework executed on a Microsoft Operating platform. While JAVA RMI is operating system independent, it is limited to the Java programming language. However RMI does support the IIOP protocol which permits RMI to talk directly to CORBA. The stub and skeleton interface must be used whether access is local or remote.

While CORBA is a high performance, language and operating system independent architecture, it is difficult to develop, implement and maintain. This level of sophistication may cause problems between customers and vendors. IIOP requires non standard ports to be opened which may violate network security standards. JAVA RMI and Microsoft's DCOM faces various interoperability limitations.

CONTRIBUTIONS OF HTTP TRANSACTIONAL ARCHITECTURES TO SOA

HTTP-like transactional architectures include: Servlets/JSP, ASP, and PHP. While this architecture is based on a web server, it does not require a browser request. Any client program can format and transmit a HTTP request to the web server, which will invoke a server application through the CGI interface. HTTP is designed to support a large number of requests in which each HTTP is includes a limited number of transactions. HTTP is designed for volume, not throughput.

While HTTP-like transactional architectures are operating system independent, each component may written in only one CGI-based language. The client application must known the name of the application (service) and the format of the HTTP request in advanced. While the HTTP response may be formatted in either HTML or XML, the client application must have knowledge of the format of the response. HTTP, HTML, and XML data streams are treated as strings. Both the client and the server application must have knowledge of the data type, but HTTP-based solutions offer no data type protection. While HTTP-based architectures are not encumbered by firewall port restrictions and are easy to develop, program modules are not compatible between languages and the client-server interface is very difficult to maintain.

COMPARISON OF SOA TO ALTERNATIVE DISTRIBUTED COMPUTING SOFTWARE ARCHITECTURES

Distributed Computing Architectures vary in the degree of interoperability of supported languages and operating systems, type of protocol access, scalability, complexity, security and discovery. In "Teach Yourself Web Services in 24 Hours" Potts and Kopak introduced a framework to compare various technologies which were similar to SOA [14]. A further review of literature suggested other criteria that may be used to compare distributive computing technologies. A more comprehensive list of comparison criteria and explicit measurement scales would be useful to both industry and academics. As a result of Potts and Kopak work comparing alternative distributive computing technologies to SOA, the following two tables were formulated.

Table 1 presents a summarized list of criteria that can be used to compare the interoperability features of various Distributed Computing Software Architectures. More specific measurement scales are introduced. Tables 2a and 2b summarize and compare the interoperability features of various Distributed Computing Software Architectures using the criteria presented in Table 1.

DISCUSSION

A review of Tables 2a and 2b indicates that Service Oriented Architectures offer the same degree of interoperability as CORBA, but with greater integration with legacy systems, lower operational and software development costs and easier integration with customers and business partners. But, SOA is still in its infancy and many of the SOA's limitations are only being addressed by proprietary fixes, e.g., IBM's WebSphere, JAVA, BEA's Weblogic Workshop, and Microsoft's Web Services. Standards such as authentication, nonrepudiation, transaction control, scalability and software testing needs to be improved. While SOA does use a well published interface through an application directory using a WSDL document, this does not mean that a client application will automatically alter its interface when the server application does.

But Tables 2a and 2b only include a comparison of the technical issues of Distributed Computing Architectures. SOA is both a software framework and an architecture. A software framework is a reusable design for a software system (or subsystem) [7]. In this aspect, SOA is the maturation of structured and modular programming. Software Frameworks are designed with the intent of facilitating software development, by allowing designers and programmers to spend more time on meeting software requirements rather than dealing with the more tedious low level details of providing a working system.

As a software architecture, SOA must also consider external visible properties of components, and the relationships between them. SOA also refers to documentation of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects [1]. In this aspect, SOA represents the maturation of interoperability, and the need to design and manage distributed interoperable systems.

TABLE 1: COMPARATIVE DISTRIBUTED COMPUTING CRITERIA

Criteria	Description
Type	Classification of Distributed Computing Architecture: • S/S - Stub Skelton • HTTP Transactional • Web Services - SOA
Language	Distributed Computing Architecture support for various programming languages. Classified as either • Language Neutral, i.e., any programming language may be used, or • Specific language limitation cited
Legacy Language Support	Ability to interface existing applications non-object-oriented languages such as Cobol, Fortran, RPG, etc. Yes, No
Protocol	TCP/IP application protocols
Message Interface	Middleware protocol used by programs to pass data between distributed applications, via some network protocol
Connection, Volume and Transactional Support	• HTTP type of connection: Persistent or Non-presistent • Volume - a estimate of transaction scalability of a scale of low, medium an high • Transaction Integrity Support (TS) or Limited Transaction Support. Limited Transaction Support - Transactional integrity protection must be provided by application program, and is not provide by the distributed architecture.
OS	Operating System Support for the Distributed Computing Architecture. • Indep. - Operating System Independent • MS Only - Limited to specific versions of the Microsoft Operating Systems
Discovery	Automatic methodology available to determine the format of the message (interface).
Security	The degree to which authentication, encryption, and non-repudiation security services is provided by the distributed architecture. Classified as • Strong - security features provided • Moderate - some security features provided • Weak - most security features not provided
Development Level	Degree of Application Program complexity and level of programmer expertise using available Integrated Development Tools (IDE) • Difficult - requires considerable application development experience in a language and knowledge of the interface • Moderate - requires moderate application development experience in a language. The knowledge of the interface is not required. • Easy - requires some application development experience in a language. The knowledge of the interface is not required.
Cost	Estimated Relative Cost of Development
Speed	Relative Speed of Development
Maturity Level	Maturity level of Distributed Technology • High - Well developed standards, more than 10 years of experience and commercial adoption • Emerging - Incomplete standards, less than 10 years of experience and commercial adoption

TABLE 2A: COMPARISON OF DISTRIBUTED COMPUTING ARCHITECTURES

Architecture	Type	Language	Legacy Language Support	Protocol	Message Interface
CORBA	S/S	Language Neutral	Yes	IIOP, SSLOP	CORBA
RMI	S/S	Java Only	No	JRMP, IIOP, SSLOP	RMI
DCOM	S/S	.NET Framework Languages only, i.e., VB, C##	No	HTTP, HTTPS	DCOM
Servlet	HTTP transactional	Java only	No	Any Protocol	None
JSP	HTTP transactional	JSP only	No	HTTP, HTTPS	None
ASP	HTTP transactional	ASP and .Net Framework	No	HTTP, HTTPS via API	None
PHP	HTTP transactional	PHP only	No	HTTP, HTTPS via API	None
SOA	Web Service	Language Neutral, including .NET and Java	Yes	HTTP, HTTPS, FTP, SMTP	WSDL

TABLE 2B: COMPARISON OF DISTRIBUTED COMPUTING ARCHITECTURES

Architecture	Connection and Transactional Support	OS	Discovery	Security	Development, Level, Cost, Speed and Maturity
CORBA	Persistent, Med to low volumes, TS	Indep	IP /Port/ No Directiory	Weak	Difficult, High Cost, Slow, Mature
RMI	Persistent, Any Volume, TS	Indep	IP /Port/ No Directory	Strong	Difficult, High Cost, Slow, Mature
DCOM	Persistent, Any Volume, TS	MS	IP/ Port/ No Directory	Strong	Moderate, High Cost, Slow, Mature
Servlet	Non Persistent, Any Volume, TS	Indep	IP (DNS) /Port No Directory	Weak	Difficult, MedHigh Cost, Slow, Mature
JSP	Non Persistent, Med to Low Volumes, Limited TS	Indep	IP (DNS)/ No Directory	Weak	Easy, Low Cost, Med, Mature
ASP	Non Persistent, Med to Low Volumes, , Limited TS	Indep	IP (DNS)/ No Directory	Weak	Easy, Low Cost, Med, Mature
PHP	Non Persistent, Med to Low Volumes, Limited TS	Indep	IP (DNS) /No Directory	Weak	Easy, Low Cost, Med, Mature
SOA	Non Persistent, Med to Low Volumes, Limited TS	Indep	UDDI Directory	Weak	Easy, Low Cost, Med, Mature

SOA AND IS CURRICULUM

The SDLC is a mainstay of IS curricula. It includes business process redesign implicitly, but with the advent of object-oriented analysis and design techniques, the focus has shifted from business processes to technology. Business process content, which was explicit to the SDLC process, has been relegated to at best an implicit inclusion.

SOA, however, includes business processes as an inherent focus of the analysis and design methodology. With the recent trend in web based object oriented software development, CIS and IS curricula have shifted more to technology oriented objects, and away from business process redesign. Only the MIS curriculum within schools of business still, for the most part, has a direct connection to business.

The concepts in this paper are taken from the technological concerns, not business processes. It appears that it is advantageous to once more include business process design as an explicit factor in the analysis process. The inclusion of SOA in the curriculum is one way in which this can be done.

RECOMMENDATIONS FOR FUTURE RESEARCH

The software development models appropriate for each of these technologies need to be further explored. The understanding and comparison of various Distributed Computing technologies is of strategic importance as industry attempts to reuse legacy applications and lower the cost of software development, while integrating business system applications between the organization and external parties. SOA provides a viable software development architecture to fulfill this need. While SOA represents an evolution of modular and object-oriented programming paradigms, implementing SOA requires a different approach to the system development life cycle and the way programming languages and operating systems are presented. The evolution of SOA requires a serious review of IS

curricula. The strategic implications of SOA in the IS curriculum can not be ignored.

REFERENCES

1. Bass, L., Clements, P, and Kazman, R. (2003). Software Architecture in Practice, Second Edition. Addison-Wesley.
2. Burd, S. (2006). Systems Architecture, Fifth Edition. Course Technology.
3. Coulouris, G., Dollimor, J., and Kindberg, T. (2001). Distributed Systems, Concepts and Designs 3rd edition. Addison Wesley.
4. Dahl, O., Dijkstra, C., Hoare, C. (1972). Structured Programming. London: Academic Press.
5. Hoffer, J., George, J., and Valacich, K. (2002). Modern Systems Analysis and Design, Third Edition. Prentice Hall.
6. ISO/IEC JTC1 (2003). Information Technology for Learning, Education, and Training International Standards Organization.
7. Johnson, R., and Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming* 1(2).
8. Kendall, K., and Kendall, J. (2002). Systems Analysis and Design, Fifth Edition. Prentice Hall.
9. Kohun, F., Wood, D., Laverty, J. (2007). Systems Oriented Architecture, Unified Process life Cycle, and IS Model Curriculum Compatibility: Meeting Industry Needs. *Information Systems Education Journal*, 5(1).
10. Malik, D. (2006). Java Programming: Program Design Including Data Structures. Course Technology.
11. Natis, Y. (2003). Service-Oriented Architecture Scenario. *Gartner Research* AV-19-6751.
12. Newcomer, E. and Lomow, G. (2005). Understanding SOA with Web Services. Addison Wesley..
13. Parnas, D. (1972). On the Criteria To Be Used in the System to Decompose Modules. *Communications of the ACM* 15(12).
14. Potts, S. and Kopack, M. (2003). Teach yourself Web Services in 24 Hours Sams Publishing.
15. Szyperski, C. (2002). Component Software: Beyond Objet-Oriented Programming, Second Edition. Addison-Wesley.
16. Tanenbaum, A. (2008). Modern Operating Systems, Third Edition. Pearson/Prentice Hall.